CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–21
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

An Implementation and Evaluation of the MPI 3.0 One-Sided Communication Interface

James Dinan,[1] Pavan Balaji,[1] Darius Buntinas,[1] David Goodell,[1]
William Gropp,[2] Rajeev Thakur[1]

[1]*Argonne National Laboratory, Argonne, IL, USA.*
[2]*University of Illinois at Urbana-Champaign, Urbana, IL, USA.*

SUMMARY

The Message Passing Interface (MPI) 3.0 standard includes a significant revision to the remote memory access (RMA) one-sided communication system expected to greatly enhance the usability and performance of MPI RMA. We present the first complete implementation of MPI-3 RMA and document implementation techniques and performance optimization opportunities enabled by the new interface. Our implementation targets messaging-based networks and is publicly available in the latest release of the MPICH MPI implementation. Using this implementation, we explore the performance impact of new MPI-3 functionality and semantics. Results indicate that the MPI-3 RMA interface provides significant advantages over the MPI-2 interface, by enabling significant increase in communication concurrency through relaxed semantics in the interface and additional routines that provide new window types, synchronization modes, and atomic operations.

Received . . .

KEY WORDS: Message Passing Interface (MPI), One-Sided Communication, Remote Memory Access (RMA), MPICH

1. INTRODUCTION

One-sided communication has become an increasingly popular and important communication paradigm, and its impact has been demonstrated through a wide-variety of computational science domains, including computational chemistry [34], bioinformatics [27], earthquake modeling [9], and cosmology [30]. Unlike traditional two-sided and collective communication models, one-sided communication decouples data movement from synchronization, eliminating overhead from unneeded synchronization and allowing for greater concurrency. In addition, message matching and buffering overheads that are required for two-sided communications are eliminated, leading to a significant reduction in communication costs.

A variety of parallel programming systems, such as those in the one-sided [4, 8, 14, 24] and Partitioned Global Address Space (PGAS) families of programming models [6, 19, 25, 26, 33], provide a one-sided communication interface to applications. The Message Passing Interface (MPI) Forum added support for one-sided communication (also known as remote memory access, or RMA) in version 2.0 of the MPI standard [22], to function alongside MPI's traditional two-sided and collective communication models. Like the rest of the MPI standard, the MPI-2 RMA model was

*Correspondence to: Argonne National Laboratory. E-mail: dinan@mcs.anl.gov

Contract/grant sponsor: U.S. Department of Energy; contract/grant number: DE-AC02-06CH11357

designed to be performant and extremely portable—even on systems that lack a coherent memory subsystem. While MPI-2 was effective for a variety of applications and systems, it lacked various communication and synchronization features, and its conservative memory model limited its ability to efficiently utilize hardware capabilities, such as cache coherence. Combined, these factors led some to conclude that MPI-2 RMA was not capable of supporting important classes of higher-level programming models [5].

The MPI Forum recently ratified version 3.0 of the MPI standard [23]. MPI-3 includes a broad update to the RMA interface that attempts to rectify the issues identified in the MPI-2 model. MPI-3 is backward compatible with MPI-2 and adds a variety of new atomic operations, synchronization primitives, window types, and a new memory model that better exposes the capabilities of architectures with coherent memory subsystems. It is believed that these features will address issues in the MPI-2 model and greatly improve the usability, versatility, and performance potential of MPI RMA.

MPICH is one of the most widely used implementations of MPI, and it has been the first fully compliant implementation of each version of the MPI standard. In November 2012, we created and released the first fully MPI-3 compliant implementation in MPICH 3.0. This included a broad renovation of the RMA infrastructure to incorporate the new functionality in MPI-3 and open new opportunities for performance-focused communication runtime research. In this paper, we present the design details of the MPICH implementation of MPI-3 RMA. We evaluate the performance impact of new MPI RMA features, including those enabled by the MPI-3 interface itself (compared with the MPI-2 RMA interface) as well as those enabled by the MPICH implementation of MPI-3 RMA. Early results indicate that MPI-3 provides significant improvements in performance over the MPI-2 RMA interface.

The presentation of our work is organized as follows. In Section 2, we discuss the MPI RMA interface, including its history and the new additions in MPI-3. The overall architecture of the MPICH implementation of MPI is presented in Section 3. In Section 4 we present various design aspects of the MPICH RMA implementation. A detailed performance evaluation of our implementation is presented in Section 5. We discuss existing research related to this effort in Section 6. In Section 7 we conclude with a discussion of plans for future work.

2. THE MPI REMOTE MEMORY ACCESS INTERFACE

The MPI remote memory access interface is a large and sometimes complex specification. The RMA interface was introduced in the MPI 2.0 standard, and it specified a conservative data consistency model, which was intended to support systems with and without cache coherence. The recent MPI 3.0 standard is backward compatible with MPI-2 and extends this model in a several significant ways, adding new functionality and optimization opportunities for cache-coherent networks and significantly expanding the set of communication and synchronization mechanisms provided by the interface.

2.1. History of MPI Remote Memory Access

The MPI-2 specification process started in March 1995 and culminated with the release of the MPI 2.0 document in July 1997. This document included the first version of the MPI RMA one-sided communications interface. The design and text of the MPI specification included contributions from several of the authors of this paper. The goals of the MPI-2 RMA interface included providing a *portable* interface for one-sided communication; separating data movement from interprocess synchronization; and supporting cache-coherent and non-cache-coherent systems.

In spite of achieving these objectives, the MPI-2 RMA interface has been found to be inadequate for many common one-sided use cases [5]. Issues with the interface include collective-only window creation, restrictions on which memory can be exposed for RMA, erroneous (rather than undefined) concurrent load/store/put/get semantics, and one-target-per-epoch restrictions.

DOI: 10.1002/cpe

Because of these insufficiencies, the MPI Forum substantially updated and revised the MPI RMA interface with the release of the MPI 3.0 standard in September 2012. This effort involved many organizations and individuals, including the authors of this paper, over the span of several years. The update focused on addressing issues that have prevented MPI RMA from providing a common, portable one-sided substrate for higher-level one-sided and global address space models, as well as adding other features that have been demonstrated to significantly benefit application developers. These features include additional communication operations that more closely match traditional shared-memory programming models, relaxed synchronization rules, lighter-weight mechanisms for controlling ordering of communication operations, and new ways to allocate and associate memory with MPI one-sided windows.

2.2. MPI-2 Remote Memory Access

MPI-3 RMA is a proper extension to MPI-2 RMA; as such it is backward compatible. In this section, we describe the aspects of the MPI-3 RMA specification that are inherited from the MPI-2 RMA specification. At its core, the MPI-2 RMA interface is composed of a set of communication operations and two data access synchronization schemes: "active target" and "passive target" synchronization.

2.2.1. One-Sided Communication Operations All MPI RMA communication operations occur in the context of a *window*. A window is composed of a group of processes, described at window-creation time by a communicator, and a contiguous region of memory at each process. The memory region at each process may differ in size and address at each process. MPI window creation is a collective, potentially synchronous operation over the input communicator that occurs via a call MPI_Win_create. In addition to a communicator, buffer pointer, and buffer size parameter, the caller must specify a "displacement unit" and an optional set of MPI_Info hints used to enable potential optimizations by the MPI implementation. Only memory that has been exposed in a window can be accessed by using one-sided communication operations.

The MPI-2 standard defines just three communication operations: MPI_Put, MPI_Get, and MPI_Accumulate. The process that invokes a communication operation is designated the *origin*, and the process in which data is accessed is designated the *target*. The origin and target may be the same process, although performing the corresponding role in each case. Origin communication buffers are described by passing a (pointer,count,datatype) triple, whereas target communication buffers are described by passing a (displacement,count,datatype) triple. This displacement value is scaled by the displacement unit value given by the target process at window creation time.

The put operation transmits data from the origin to the target. The get operation transmits data from the target to the origin. The accumulate operation transmits data from the origin to the target, then applies a predefined MPI reduction operator to reduce that data into the described buffer at the target. The target and origin datatypes used in the accumulate operation may be derived datatypes, but they must be composed of only one distinct basic element type.

Each of these communication operations must occur in the context of either an *active target* synchronization epoch or a *passive target* synchronization epoch. In MPI RMA, all communication operations are nonblocking and are completed at the end of the synchronization epoch.

2.2.2. The MPI-2 Data Consistency Model MPI-2 RMA defined the "separate" memory model, which specifies the consistency semantics of accesses to data exposed in an RMA window. This model was designed to be extremely portable, even to systems without a coherent memory subsystem. In this model, the programmer assumes that the MPI implementation may need to maintain two copies of the exposed buffer in order to facilitate both remote and local updates on noncoherent systems. The remotely accessible version of the window is referred to as the *public* copy, and the locally accessible version is referred to as the *private* copy.

When a window synchronization is performed, the MPI implementation must synchronize the public and private window copies. Thus, MPI-2 forbids concurrent overlapping operations when any of the operations writes to the window data; the only exception is that multiple accumulate

Prepared using cpeauth.cls
DOI: 10.1002/cpe

operations can perform concurrent overlapping updates when the same operation is used. Because communication operations are nonblocking, the programmer must ensure that operations performed within the same synchronization epoch do not perform conflicting accesses. In addition, because the MPI library is unaware of which locations are updated when the window buffer is directly accessed by the hosting process, local updates cannot be performed concurrently with any other operations. Any violation of these semantics is defined to be an MPI error.

2.2.3. Active Target Synchronization In the active target mode in MPI-2, data is transmitted from the memory of one process to the memory of another, with direct participation from both processes.

The simplest form of active target synchronization uses the concept of a "fence." All processes in the window collectively call MPI_Win_fence in order to demarcate the beginning and end of RMA epochs. During these epochs the application may issue zero or more MPI communication operations or in some cases may perform direct load/store operations to that process's portion of the window. RMA operations issued before the fence call began will be completed before the call returns.

The simplicity offered by fences comes at a cost to application flexibility. To address this, MPI offers a more versatile, more complex synchronization mode known as generalized active target synchronization (GATS). This facility is sometimes also known as post/start/complete/wait (PSCW), referring to the primary synchronization routines involved in this mode. This mode differentiates between two different epoch types: an *exposure epoch* and an *access epoch*. Processes that will be accessed as targets must invoke MPI_Win_post calls, supplying a group argument to indicate the set of peer processes that will perform communication operations targeting the posting process. Those origin processes must all correspondingly invoke MPI_Win_start to begin an access epoch before issuing any communication operations. The start routine also takes a group argument indicating with which target processes the calling process will communicate. When the accessing processes have posted all RMA operations, they must call MPI_Win_complete to end the access epoch and force origin completion of the previously issued communication operations. Target processes call MPI_Win_wait (or MPI_Win_test repeatedly) in order to wait until any communication operations initiated during the exposure epoch are completed at the target (this process).

GATS makes synchronization more flexible by making it possible to limit the number of processes with which any given process must synchronize to a minimal subset (e.g., neighboring processes in a halo exchange operation). Only processes that actually communicate with each other must synchronize. Furthermore, the synchronization pattern may be asymmetric in some ways because of the separation of the exposure epoch from the access epoch.

2.2.4. Passive Target Synchronization For applications where synchronization should be avoided or the communication pattern is difficult to predict, MPI offers an alternative to active target synchronization, known as passive target synchronization. Processes perform communication operations in access epochs demarcated by MPI_Win_lock and MPI_Win_unlock calls. Despite the names, these routines do not provide a traditional lock or mutex. Instead they serve two primary purposes: (1) to group one-sided communication operations and certain load/store access that target a particular process, and (2) to ensure completion (visibility) of specific accesses relative to other accesses.

These purposes are compatible with a lock/unlock implementation based on strict mutual exclusion. However, the lock routine is not required to synchronously block in order to acquire such a lock and may instead synchronize in the background or defer synchronization and communication altogether until the synchronizing unlock.

An RMA lock may be either *exclusive* or *shared*. Performing accesses in an exclusive lock epoch ensures that no other lock/unlock epochs (of either type) will appear to occur concurrently. A shared lock allows multiple origin processes to access the window at the target concurrently. If both shared and exclusive epochs are requested, MPI ensures mutual exclusion of shared and exclusive epochs. The application is responsible for ensuring that accesses in a shared lock epoch do not conflict with accesses from concurrent shared lock epochs originated by other processes.

DOI: 10.1002/cpe

2.3. MPI-3 Remote Memory Access Extensions

The MPI-3 standard greatly enhanced the MPI-2 RMA functionality, specifically with improvements for atomic operations, finer-grained control over operation completion and synchronization in passive target epochs, relaxed access restrictions when hardware-assisted coherence is available, and new window types that enable dynamic exposure of memory for RMA and interprocess shared memory.

2.3.1. One-Sided Atomic Operations Three new atomic operations were added in MPI-3 RMA: MPI_Get_accumulate, MPI_Fetch_and_op, and MPI_Compare_and_swap. Get-accumulate and fetch-and-op both provide atomic read-and-update operations; get-accumulate is general purpose and allows the programmer to provide derived datatypes and differing communication parameters for each buffer, whereas fetch-and-op restricts its use to a single element of a predefined MPI datatype. Because of these restrictions, fetch-and-op offers numerous optimization opportunities to the MPI implementation, potentially reducing software overhead latencies and permitting direct use of hardware-supported atomic operations.

Compare-and-swap atomically compares a "compare buffer" at the origin with the target buffer and replaces the target buffer contents with the (separate) origin data buffer contents if the values are equal. The original value of the target buffer is always returned to the caller at the origin. This operation is limited to the integer subset of predefined datatypes, and the same datatype must be used for all buffers.

All three of these new operations are safe to use concurrently with each other and the MPI-2 accumulate operation. Atomicity for accumulate and get-accumulate operations with derived datatypes or a count greater than one with predefined datatype occurs elementwise, at the granularity of basic, predefined MPI datatypes. The atomic and accumulate operations are *not* atomic with respect to put and get operations. Instead, accumulate with the MPI_REPLACE operation may be used as an atomic "put," and get-accumulate with the MPI_NO_OP operation may be used as an atomic "get."

2.3.2. Request-Generating Operations All MPI RMA communication operations are nonblocking, but MPI-2 operations do not return a request handle. Instead, completion is managed through synchronization operations such as fence, PSCW, and lock/unlock. MPI-3 adds "R" versions of most of the communication operations that return request handles, such as MPI_Rput and MPI_Raccumulate. In turn, these requests can be passed to the usual MPI request completion routines, such as MPI_Wait, to ensure local completion of the operation. This provides an alternative mechanism for controlling operation completion with fine granularity. However, these request-generating operations may be used only in passive-target synchronization epochs (i.e., with lock/unlock).

2.3.3. MPI-3 RMA Windows MPI-3 adds three new window types: MPI-allocated windows, dynamic windows, and shared-memory windows. MPI-allocated windows are created by calling MPI_Win_allocate. In contrast to MPI-2 windows where the user supplied the window buffer, MPI allocates the buffer for this window, enabling the MPI implementation to utilize special memory (e.g., from a symmetric heap or a shared segment) or optimize the mapping for locality.

All MPI windows are created collectively over the input communicator, and traditionally each process could associate only one contiguous region of memory with the window. This restriction posed significant challenges to applications that need to dynamically allocate and deallocate memory. MPI-3 addresses this issue by providing a new *dynamic window* facility, which collectively creates a window with no initially associated memory. Memory can then be asynchronously attached to (or detached from) this window by individual processes. The new routines MPI_Win_create_dynamic, MPI_Win_attach, and MPI_Win_detach facilitate this new functionality.

Shared memory programming can provide an efficient means for utilizing on-node resources [15]. To this end, many programmers combine MPI with OpenMP or a threading library, which adds

Prepared using cpeauth.cls
DOI: 10.1002/cpe

	Load	Store	Get	Put	Acc
Load	OV+NOV	OV+NOV	OV+NOV	NOV	NOV
Store	OV+NOV	OV+NOV	NOV	X	X
Get	OV+NOV	NOV	OV+NOV	NOV	NOV
Put	NOV	X	NOV	NOV	NOV
Acc	NOV	X	NOV	NOV	OV+NOV

(a) Separate Model

	Load	Store	Get	Put	Acc
Load	OV+NOV	OV+NOV	OV+NOV	NOV+BOV	NOV+BOV
Store	OV+NOV	OV+NOV	NOV	NOV	NOV
Get	OV+NOV	NOV	OV+NOV	NOV	NOV
Put	NOV+BOV	NOV	NOV	NOV	NOV
Acc	NOV+BOV	NOV	NOV	NOV	OV+NOV

(b) Unified Model

Figure 1. Valid concurrent (possibly overlapping) operations in the "separate" and "unified" memory models. OV: overlapping permitted. NOV: only nonoverlapping permitted. BOV: overlapping permitted at single-byte granularity. X: not permitted.

the complexity of managing two parallel programming systems in the same program. MPI-3 adds a new shared-memory window, created via MPI_Win_allocate_shared, which allows processes to portably allocate a shared-memory segment that is mapped into the address space of all processes participating in the window. In addition, the new MPI_Win_sync routine provides a load/store fence that is lighter weight than does a full window synchronization. By using MPI RMA synchronization and atomic operations, shared-memory windows provide programmers with a complete, portable, interprocess shared-memory programming system.

2.3.4. Unified Memory Model No coherence in the memory subsystem or network interface is assumed by the MPI-2 RMA "separate" memory model, resulting in logically distinct "public" and "private" copies of the window copies described in Section 2.2.2. This conservative model is a poor match for computers with coherent memory subsystems, as it does not provide access to the system's full performance and programmer productivity potential. A new "unified" memory model was added in MPI-3 to address this shortcoming.

The unified model relaxes several restrictions present in the separate model by assuming that the public and private copies of the window are logically the same memory. These restrictions and relaxations are summarized in Figure 1. This figure shows which operations are permitted to occur concurrently in the same window and whether those concurrent operations are permitted to access overlapping regions of the window.

Support for the unified model is not required by the MPI-3 standard. Users must query the implementation via the predefined MPI_WIN_MODEL attribute in order to ensure that a particular window supports the unified model before taking advantage of its relaxed consistency semantics.

2.3.5. Passive Target Synchronization New synchronization mechanisms were included in the MPI-3 passive target synchronization mode. MPI-2 passive target mode was restricted to a simple lock/unlock interface, where an origin process could lock only one target at a time. MPI-3 permits locking multiple targets simultaneously. It also offers a new MPI_Win_lock_all routine that is equivalent to locking each target in window with a shared lock.

When using lock-all, finer-grained synchronization can be achieved with the request-generating operations discussed in Section 2.3.2. It can also be achieved with the new "flush" routines: MPI_Win_flush and MPI_Win_flush_local. These routines specify a particular target and ensure that all operations initiated to that target before the flush are remotely complete at the target (in the case of flush) or locally complete at the origin (in the case of flush-local). These routines also have "all" variants, MPI_Win_flush_all and MPI_Win_flush_local_all, that are equivalent to flushing each target in the window in sequence.

3. ARCHITECTURE OF MPICH

MPICH is an implementation of the MPI standard developed at Argonne National Laboratory. A primary goal of the project is to provide a portable high-performance implementation that can be ported and adapted as necessary by third-party developers to support various architectures

DOI: 10.1002/cpe

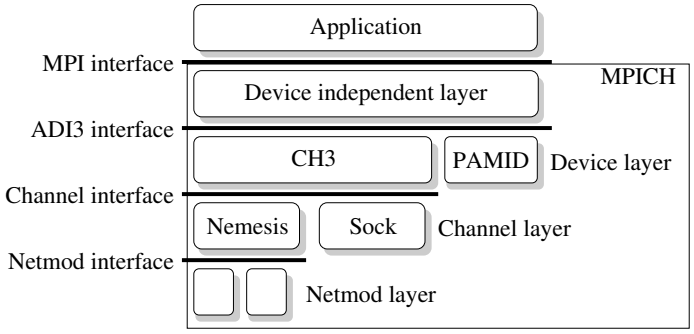


Figure 2. Internal interfaces and layers of MPICH.

and interconnects. For example, IBM and Cray have ported and adapted MPICH to support their supercomputers. To realize this goal, MPICH is designed with various internal portability interfaces allowing third-party developers the flexibility to choose the best interface when porting MPICH for their system. Figure 2 shows these interfaces and the internal layers of MPICH. The top layer in the figure is the application that uses the MPI interface to communicate with MPICH. Below this layer is the *device-independent layer*. This layer implements functionality such as object management and error handler management that would be common to all derivatives of MPICH. This layer also implements collective communication operations, which can be overridden by lower layers if needed to provide implementations that are optimized to a particular platform. The device-independent layer exposes the *ADI3* interface to the *device layer* below it. A developer has the option to implement a *device* at this layer to support a particular platform. For example, the PAMID device supports platforms using the PAMI interface, such as IBM's Blue Gene/Q.

The ADI3 interface provides the greatest flexibility for the developer; however, implementing a device using this interface also requires considerable effort because of the number of functions that must be implemented. For this reason MPICH has a default device called *CH3*. The CH3 device implements functionality such as message matching, connection management, and handling of one-sided communication operations and exposes a significantly simpler interface called the *channel interface*. Developers can choose to implement a *channel* to support their platform. The figure shows two channels: *Nemesis* and *Sock*. Although there is an additional layer between a channel and the application as compared to a device, common cases are fast-pathed through CH3 by using function pointers or function inlining, so the overhead of the additional layer is minimized and in some cases avoided entirely.

The Sock channel uses TCP exclusively for communication, whereas the Nemesis channel uses shared memory for intranode communication and a network for internode communication. Nemesis exposes the *netmod* interface that allows the developer to implement a network module to support a particular interconnect. As with CH3, although Nemesis adds an additional logical layer between the network module and the application, common cases are fast-pathed to avoid performance overhead. In fact, the Nemesis TCP network module outperforms the Sock channel despite the additional logical layer.

We implemented our messaging-based one-sided implementation in CH3 because (1) at this layer, all channels and network modules will be able to use the one-sided operations without having to reimplement them; (2) functionality that is needed by the one-sided implementation, such as processing of MPI datatypes and handling of packets, is available in CH3 but not at a higher level; and (3) an implementation of MPI 2 one-sided operations has previously been implemented in CH3, so we can reuse some of the existing functionality. Next we briefly describe three aspects of the CH3 architecture that are relevant to the implementation of messaging-based one-sided operations: datatype processing, sending and receiving messages, and allocating and attaching to shared memory regions.

DOI: 10.1002/cpe

3.1. Datatype Processing

MPI *datatypes* are descriptions of the layout of data in memory. A simple datatype may describe a contiguous buffer, whereas a more complicated datatype may describe a section of a multidimensional matrix. While the datatypes are defined by the application in a recursive manner, CH3 processes the datatypes in a iterative manner to improve performance. In MPI a buffer is described by a pointer, a datatype and a *count* specifying the number times the datatype repeats. CH3 internally defines a *segment* object, which consists of a buffer pointer, datatype and count tuple to identify the data to by sent or received, as well as an offset specifying a location in the stream of bytes defined by the tuple. Once a segment is constructed, it can be passed to various datatype processing functions either to pack the data into or unpack the data from a contiguous buffer or to generate a different representation of the buffer (e.g., an I/O vector).

3.2. Sending and Receiving Messages

While the channel is responsible for actually putting bytes into and pulling bytes out of the network, CH3 assembles packets for sending and processes received packets. To send a packet, CH3 creates the packet header, then calls a channel send function passing a pointer to the header and a description of the data to be sent. If the data is contiguous, the description is simply a pointer to the buffer and the size. For noncontiguous data the description consists of a datatype *segment*. The channel uses the datatype-processing engine either to convert the description into a form usable by the underlying network (e.g., an I/O vector) or to pack the data into contiguous buffers. The send functions are nonblocking and therefore need to queue packets that cannot be immediately sent.

CH3 uses packet handler functions to process incoming packets. Once an entire header has been received from the network, the packet handler function is determined based on the packet type and is called with a pointer to the buffer where the packet is stored along with the number of bytes that have been received. The channel does not need to know the size of the packet, only the size of the header. If the buffer contains the entire packet, the handler will process the packet and then return the size of that packet. If the buffer does not contain the entire packet, the handler will process as much of the packet as possible and then return a request indicating where the channel should put the remainder of the packet when it is received, along with a handler function to call when the data has been received.

3.3. Allocating and Attaching to Shared Memory

CH3 provides functionality for allocating and attaching to shared-memory regions. The shared-memory allocation function takes the size as an input parameter and returns a pointer to the allocated region along with the a handle. The handle can be *serialized* into a character string that can then be communicated to other processes. The serialized handle can be sent by using CH3 messages or using an out-of-band communication mechanism such as the process management interface (PMI) [1]. When the serialized handle is received by another process, it can be used to attach to the shared-memory region allocated by the other process.

4. DESIGN OF MPI-3 RMA IN MPICH

We have extended the MPICH RMA implementation with support for new MPI-3 RMA functionality. Our implementation is integrated in CH3 and can be used with a variety of networks that are supported as CH3 channels and Nemesis network modules, as described in Section 3. The MPI-2 RMA design focused on a messaging-based design, and we have developed MPI-3 RMA within this design space; in the near future, we plan to extend this design to leverage one-sided network capabilities.

DOI: 10.1002/cpe

4.1. MPI-3 RMA Windows

In the MPICH RMA design, a window object contains base pointers and displacement units for all processes in the window's group. When an RMA operation is issued, the origin process calculates the effective address at the target process and transmits this in the packet header. This approach reduces the work that the target must do and in some cases, avoids a window object lookup at the target. However, the use of an $O(P)$ structure can limit scalability. In the future, we will investigate other approaches, such as transmitting the displacement instead of the effective address, in order to improve scalability.

MPI-3 defines several new RMA window types, referred to as window *flavors*: MPI-allocated windows, dynamically allocated windows, and MPI-allocated shared memory windows. MPI-allocated windows allow MPI to map the memory that will be targeted by RMA operations, potentially enhancing performance. In CH3, MPI-allocated windows allocate memory using the device's memory allocator, which can be used to allocate memory that is associated with the device.

Dynamically allocated windows enable a powerful new usage pattern where memory can be asynchronously attached to and detached from the window by the origin process. The address of the memory is used directly as the window displacement argument to RMA operations, obviating the need for $O(P)$ structures. In the MPICH design, we expose all of memory in every dynamic window, effectively making attach and detach operations no-ops. One could validate that operations target only attached memory; however, this check has a significant performance penalty and is not required by the MPI standard.

4.2. Window Synchronization

MPICH maintains RMA operation queues, which are flushed when an access epoch is completed [31]. When a given window is in active target mode, all operations are batched in a single queue, since the completion of any active target access epoch will require the completion of all operations. When a window is accessed by using passive target mode, individual operation queues are created to manage communication with each target. We distinguish active and passive target access epochs through the state tracking discussed in Section 4.4. This permits more efficient synchronization with individual targets, for example, flushing or unlocking an individual target process.

MPI-3 introduced several significant changes to passive target synchronization, including new flush and lock-all operations, as well as the ability to perform passive target epochs at multiple target processes concurrently. Active target synchronization was unchanged in the MPI-3 specification, and the existing MPI-2 design has changed very little. In addition to adding support for per target operation queues, the MPICH MPI-3 implementation separates release of a passive target lock from completion of operations in order to facilitate flush and request-generating operations.

Passive target lock and lock-all operations utilize the same locking facility in the MPICH design. Lock-all is a one-sided operation that requests a shared lock, and it must be compatible with (possibly exclusive mode) lock operations issued by other processes. Rather than performing a nonscalable lock operation all targets, the lock-all implementation relies on the synchronization state tracking (see Section 4.4) to indicate that all processes can be targeted by RMA operations. We track the synchronization state of each target and issue a lock request on the first synchronization with the given process.

4.3. Implementation of Communication Operations

The MPICH RMA implementation uses the message-processing capabilities to perform RMA operations in the target process' address space. We define new CH3 packet types, packet handlers, and request handlers for each new MPI-3 RMA operation. When performing an operation, the origin process generates a packet header with the corresponding operation type; populates the header with the communication parameters; and sends the header, any datatypes, and the data payload to the target. When the packet header is received by the target, CH3 dispatches the packet handler

DOI: 10.1002/cpe

corresponding the packet type field in the message header, and the packet handler performs the operation at target and sends a response to the origin, if needed.

If a derived datatype is specified for the target, an additional step is required to transmit the serialized datatype, whereas predefined datatypes are compact and can be embedded in the packet header. The derived datatype is serialized at the origin and transmitted following the packet header; the size of the derived datatype is contained within the packet header and used by the target to determine when the full datatype has arrived. Once the target has processed the packet header, it may need to wait for the arrival of the datatype. To facilitate this process, it generates a request and registers the operation's request handler which the CH3 progress engine will use to continue the RMA operation when additional datatype data has arrived.

The receipt of user data is also facilitated by the progress engine, through the use of requests. For put and accumulate operations, once the datatype information has arrived in the data stream, the remaining data forms the data payload and it is received and processed by using an additional request handler. For get operations, the origin transmits a reference to its receive request in the packet header. This reference is included in the target's response, and is used to match the get operation with the corresponding local request that contains the origin's communication parameters.

The new MPI_Fetch_and_op and MPI_Compare_and_swap operations restrict the datatypes that can be used to ensure higher performance. For these operations, the RMA communication protocol is simplified significantly. Because fewer communication parameters are needed, surplus space is available in the packet header. We use this space to embed the origin data and avoid additional steps required to transmit the payload data.

MPI-3 also introduced request-generating operations, which return a request to the user that can be used to wait for completion of a specific RMA operation. In the MPICH implementation, we use the MPICH extended generalized request framework to support these operations. We enqueue request-generating operations in the corresponding RMA operations queue and return a request handle that contains a reference to the window. When the user completes the request, we perform a local flush of the window to the target process. We plan to improve this design by enabling completion of only the operation corresponding to the request.

4.4. Efficient Synchronization State Tracking and Error Detection

MPI-3 added several refinements to passive target synchronization, including a lock-all passive target communication mode, request-generating operations, and flush operations. In addition, MPI-2 allowed only one passive target epoch at a time using lock/unlock operations; MPI-3 has lifted this restriction and allows a process to initiate one passive target epoch to every process in the window's group.

We have redesigned the RMA error detection in MPICH to detect incorrect use of RMA synchronization operations. An important design goal was that error checking add no more than several tens of cycles of overhead. To achieve this, we took a state machine approach to defining correct use of RMA synchronization calls at each process. The corresponding state transition diagram we developed is shown in Figure 3. This diagram captures all correct uses of MPI calls; any deviation is erroneous and is reported by the MPI implementation. Examples of incorrect usage include unmatched lock/unlock calls, mixing of passive and active target synchronization, use of flush or request-generating operations in active target, and mixing of passive target lock and lock-all synchronization.

As shown in the diagram, fence operations require additional state to track. The *fence_called* state changes only during collective calls to fence. If fence has been called without the NOSUCCEED assertion, it is possible to enter into a fenced active target access epoch. However, it is also valid to ignore the call to fence (i.e., it may have closed an active target phase in the program) and perform a different RMA access type. It is invalid to perform a fenced active target access epoch if another synchronization mode is used on the window; however, we do not currently detect this error because of the state tracking complexity.

DOI: 10.1002/cpe

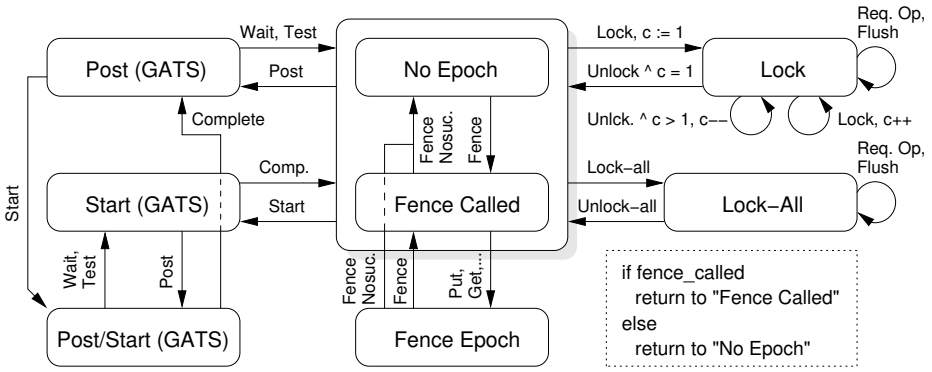


Figure 3. RMA synchronization state tracking diagram. Dashed lines indicate that a particular state is bypassed, depending on the fence state of the process.

4.5. Shared-Memory Windows

The unified memory model introduced in MPI 3 allows for efficient one-sided operations on systems with coherent memory. In [15] we described the design and implementation of shared memory windows that can use the unified memory model. In this section we briefly summarize the design and implementation of shared-memory windows in MPICH.

When a shared-memory window is created, the processes perform an all-gather operation to collect the sizes of the individual window segments of each process. The total size of the window is computed, and the root process (the process with rank 0 in the communicator associated with the window) allocates a shared memory segment of that size as described in Section 3.3. The root then broadcasts the serialized window handle, and the other processes attach to the shared-memory region. Because each process knows the size of every other process's window segment, each process can compute the offset into the shared-memory region where any process's window segment begins. The pointer to the beginning of each process's window segment is stored in an array.

If alloc_shared_noncontig info argument is specified when the window is created, the individual process's window segments are not required to be contiguously arranged, and the implementation is free to allocate them in a more optimal manner; for example, each window segment may be aligned on a page boundary. In MPICH, rather than allocating a separate shared-memory region for each window segment, a single shared-memory region is allocated as before, except that the size of each window segment is rounded up to the next window size. In this way only a single shared-memory region needs to be created and attached, but each window segment is aligned on a page boundary.

Put and get operations are implemented by having the origin process directly access the memory of the target process in the shared memory region. Accumulate operations are required to be atomic, so an interprocess mutex is created for each window to serialize accumulate and atomic operations. Because processes are directly accessing the memory of other processes, window synchronization operations include appropriate memory barriers to ensure the proper ordering of memory accesses between the processes.

5. EXPERIMENTAL EVALUATION

We use our implementation of MPI-3 RMA to evaluate the performance impact of new functionality and semantic changes introduced in MPI-3. Our RMA implementation has been integrated into MPICH 3.0.1 and is publicly available. We evaluate the effectiveness of several major changes in the MPI RMA specification: the new unified memory model, new atomic communication operations, new passive target synchronization operations, and new window types.

We conducted our experiments on the Eureka cluster at the Argonne Leadership Computing Facility. This cluster is configured with 100 nodes, each with two quad-core Intel Xeon processors

DOI: 10.1002/cpe

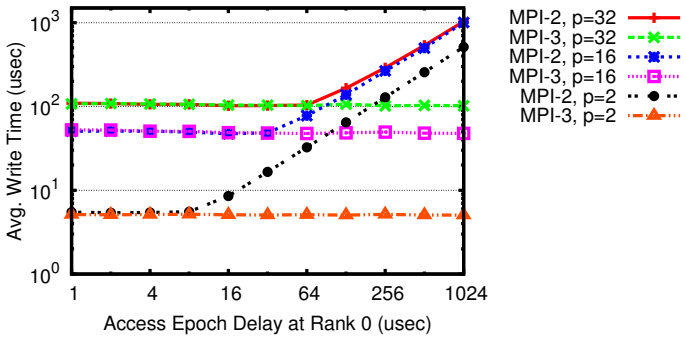


Figure 4. Average time for remote processes to write to process 0, which directly accesses the window buffer. The direct access interval length at rank 0 is varied, and timings are shown for the separate (MPI-2) and unified (MPI-3) memory models, for 2 (2 nodes), 16 (2 nodes), and 32 (4 nodes) processes.

and 32 GB of memory. Nodes are connected by using Myricom 10 Gb/s CX4 Myrinet network interfaces, configured with a 5-stage Clos topology. MPICH was configured to use the Myrinet MX network module.

5.1. Impact of the Unified Memory Model

MPI-3 RMA defines the unified memory model, which relaxes window access semantics for systems where hardware can provide the needed level of data consistency. A key difference between the unified and separate memory models is that the unified model permits RMA operations concurrent with nonoverlapping load/store operations.

We measure the performance impact of the unified memory model through a simple benchmark where rank 0 directly accesses its window data while other processes are attempting to write to a nonoverlapping location in rank 0's window. This scenario frequently arises in applications that operate directly on data exposed in a window, for example, by performing a DGEMM operation on a matrix that is exposed for RMA. In the separate (MPI-2) memory model, rank 0 must use an exclusive access epoch in order to avoid conflict between its load/store operations and remote RMA operations; in MPI-3, the unified model enables all processes to use shared access epochs.

We show the result of this benchmark in Figure 4. The figure shows the average time to write data in the separate (MPI-2) and unified (MPI-3) memory models. With MPI-2, since exclusive locks are used, operations from different processes are serialized causing the average write time to increase fairly linearly with computational cost. With MPI-3, since shared locks can be used, operations are not delayed by rank 0's direct access interval, and the average write time stays fairly constant.

5.2. Impact of Atomic Operations

MPI-3 introduced several new atomic operations, including compare-and-swap, fetch-and-op, and get-accumulate. These operations greatly increase the capabilities of MPI, especially in the construction of higher-level synchronization operations. A significant limitation in MPI-2 was that the construction of a high-level mutex library required the use of nonscalable data structures [11, 28]. Mutexes are required by a variety of higher-level one-sided libraries, including shared file pointer I/O [18, 28] and PGAS models [10]. In this experiment, we demonstrate the use of MPI-3 atomic operations through the creation of a mutex library that uses the MCS algorithm [20], which provides better scalability and significant performance gains in the presence of lock contention.

5.2.1. MPI-2 Mutex Algorithm The best-known mutex algorithm for MPI-2 RMA [28] uses a byte vector B of length $nproc$ located on the process hosting the mutex; the ith entry in this vector indicates whether process i has requested the lock. Initially $B[0 \ldots nproc - 1] = 0$. A lock operation from process i performs several nonoverlapping communication operations in a single

DOI: 10.1002/cpe

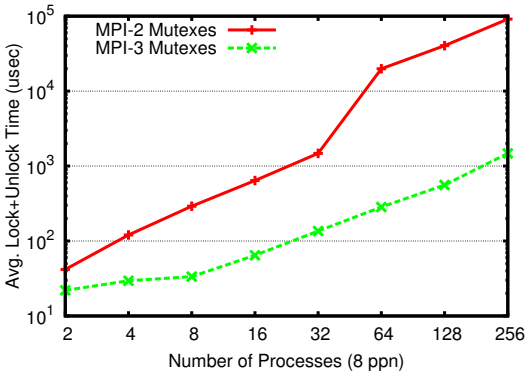


Figure 5. Average time per lock-unlock cycle for a mutex on rank 0, requested by all processes 10,000 times.

MPI RMA exclusive access epoch: entry $B[i]$ is set to 1, and all other entries are fetched. If all other entries are 0, the lock operation has succeeded; otherwise the lock operation has effectively enqueued process i in the waiting queue for the mutex. Once enqueued, the process waits on an MPI_Recv operation from a wildcard source.

When process i performs an unlock operation, it again performs an exclusive RMA access epoch on B that sets $B[i] = 0$ and fetches all other entries. B is then scanned for an enqueued request starting at entry $i + 1$, which ensures fairness. If a request is found in the queue, a zero-byte notification message is sent to this process, forwarding the lock. If no request is found, the unlock is finished.

This algorithm has two significant drawbacks. It allocates an array of size $O(P)$ bytes at the process that hosts the mutex. In addition, it requires the use of an exclusive access epoch in order to access the mutex structure.

5.2.2. MPI-3 Mutex Algorithm We have created a new mutex library, which uses the MCS algorithm [20]. Like the MPI-2 algorithm, this is also a fair, queueing mutex. Rather than storing the queue on a single process, as was the case in the MPI-2 algorithm, the MCS algorithm creates a distributed linked queue. A shared tail pointer is created on the process that hosts the mutex and all processes allocate one list element, which holds the *next* pointer, an integer value that indicates the next process waiting on the mutex.

Initially, the tail pointer contains -1, indicating that the mutex is available. When processes request the mutex, they perform an atomic swap to exchange their rank for the value in the tail pointer. If the swap operation returns -1 the process has obtained the mutex; otherwise it is the new tail of the list and it updates the *next* pointer of the previous tail process's list element. In order to ensure data consistency, list elements are updated by using the fetch-and-op operation, where the MPI_REPLACE operation is used to write to the shared location and the MPI_NO_OP operation is used to read it. Once they have updated their ancestor's *next* pointer, enqueued processes wait in an MPI_Recv operation for the previous process to forward the mutex.

When releasing the mutex, the last process must reset the tail pointer to -1. An atomic compare-and-swap is performed on the tail pointer; if the process releasing the mutex is the tail, it replaces the old tail pointer with -1. Otherwise, the process polls waits for its *next* pointer to be updated and forwards the mutex to the next process.

In contrast with the MPI-2 algorithm, the MCS algorithm in MPI-3 uses a distributed queue, which space for one integer value per process. In addition, it utilizes new atomic compare-and-swap, update, and read operations that enable processes to use the shared lock access mode.

5.2.3. Performance Comparison Figure 5 shows the average time to acquire and release the mutex in the MPI-2 and MPI-3 mutex libraries. A single mutex is created, and all processes perform 10,000 lock and unlock operations. While both mutexes use a queueing algorithm that does not

DOI: 10.1002/cpe

```
Lock(rank)
Put(write_data, rank)
Unlock(rank)

Lock(rank)
Get(read_data, rank)
Unlock(rank)
```

(a) Ordering by epoch

```
Lock_all()

...

Accum(write_data, rank, REPLACE)
Get_accum(read_data, rank, NO_OP)
Flush(rank)
```

(b) Ordered accumulate operations

Figure 6. Pseudocode for the location consistent read-after-write benchmark. In MPI-2, separate epochs are needed to order operations; in MPI-3, ordering is provided for accumulate operations.

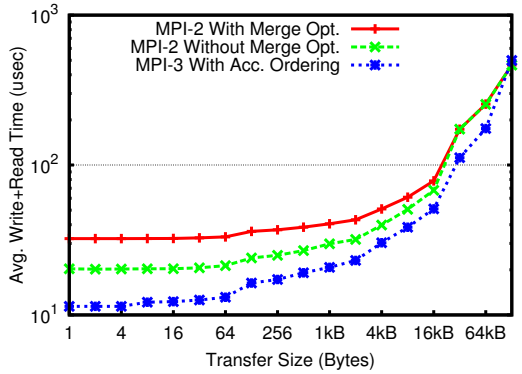


Figure 7. Average time per write followed by read between processes on two nodes, using MPI-2 with and without the lock-op-unlock optimization, and MPI-3 with ordered accumulate operations.

poll over the network, the MPI-3 MCS algorithm uses a shared lock that enables a greater amount of concurrency and better tolerance of contention. For smaller process counts, there is roughly an order of magnitude difference in lock-unlock latency, which grows to nearly two orders of magnitude with 256 processes.

5.3. Impact of Accumulate Operation Ordering

MPI-3 introduced an optional capability that orders accumulate operations issued by a particular origin process, at each target. This has a direct impact on programming models and algorithms that require location consistency—that a given process observes the results of its own operations in the order in which they were issued. Models such as ARMCI [24] and GA [25] utilize this consistency model. This model is believed to be convenient because it resembles shared memory programming and is easier to utilize at an application level.

In Figure 6 we show pseudocode for a simple benchmark that performs a write operation followed by a read to the same process. Location consistency can be achieved by using individual epochs (the approach needed in MPI-2) or by using ordered accumulate operations. Ordering for accumulate operations is selected by using an info argument when the window is created; it is not shown in the example.

Figure 7 shows the latency for location consistent read-after-write between two processes using the ordered epochs (MPI-2) and ordered accumulates (MPI-3) approaches. MPICH includes an optimization that merges the lock, RMA operation, and unlock operations into a single one-way communication for writes and a single round-trip communication for reads [31]. We show the MPI-2 implementation with and without this optimization; when the optimization is disabled, the protocols shown in Figures 12a and 12d are used. From this data, we see that accumulate ordering results in a significant reduction in latency, because of the reduction in RMA synchronization overheads.

DOI: 10.1002/cpe

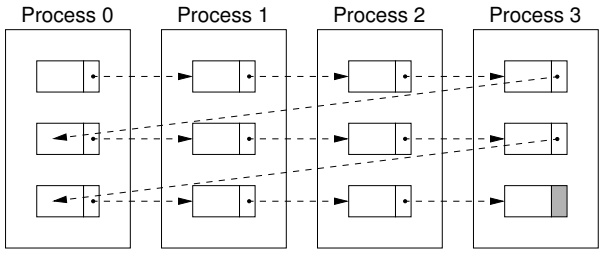


Figure 8. Structure of the linked list created by the dynamic linked list construction benchmark.

5.4. Impact of Synchronization Operations

MPI-3 has introduced new synchronization operations for passive target communication, including lock-all and flush operations. These operations provide lighter-weight synchronization than MPI-2 lock/unlock epochs do. In addition, MPI-3 accumulate operations allow concurrent overlap of accumulate calls with get-accumulate calls that perform a no-op (i.e., atomic read). This allows applications to perform concurrent reads and writes to overlapping locations in the window, with well-defined results. The combination of these new semantics and operations allows applications to express many algorithms by using MPI shared locks, which greatly increases the concurrency with which data can be accessed.

5.4.1. Dynamic Linked List Construction Benchmark We have created a linked list construction benchmark that can benefit from several of these new MPI-3 semantics. This benchmark uses a dynamic window; processes dynamically create new list elements, attach them to the window, and append them to the list. Creation of such a dynamic, distributed data structure is not possible in MPI-2, because windows are fixed in size and must be created collectively. Rather than storing the tail pointer in a fixed location, all processes traverse the list to locate the tail. Thus, this benchmark captures an application behavior where processes traverse a dynamically growing linked list—for example, a work list in a producer-consumer computation.

In order to produce a deterministic result, each process p appends a new element only when the tail pointer points to an element at process $p-1$, as shown in Figure 8. This process repeats until each process has appended N elements. Initially, process 0 creates the head of the list and broadcasts the pointer to all other processes. Pointers in MPI dynamic windows are represented by using the tuple $\langle rank,\ displacement\rangle$, and we use a rank of -1 to indicate a NULL next element pointer. When the next element pointer of the current list item is NULL, processes poll on the location of the next pointer until it is updated.

The dynamic linked list construction benchmark can be expressed by using three different synchronization idioms that utilize varying degrees of MPI-3 synchronization. Pseudocode for each idiom's linked-list traverse-and-append loop is shown in Figure 9. The exclusive lock/unlock implementation uses only MPI-2 communication and synchronization operations (but uses MPI-3 dynamic windows). The shared lock/unlock implementation uses MPI-3 accumulate operations to enable the use of MPI-2 shared-mode locks. The lock-all implementation adds the use of flush to complete communication and avoid repeated calls to lock.

5.4.2. Performance Evaluation In Figure 10 ee compare the performance of the MPI-2 and MPI-3 implementations of the linked-list construction benchmark. A key factor in the performance of this benchmark is the ability to deal with high amounts of reader-writer contention as each element is appended to the list. If we assume first-come, first-served processing of RMA operations at target processes, each writer that appends an element to the list must wait for an average of $P/2$ readers to complete polling read operations before the write succeeds. For a list of N elements and a communication latency of L, the expected execution time is $O(N \cdot P/2 \cdot L)$. Under high amounts of contention, the latency of read and write operations L increases proportional to the number of processes, P. Thus, the expected execution time is $O(N \cdot P^2)$.

DOI: 10.1002/cpe

```
for (;;) {
  if (t.rank == p-1) {
    Lock(EXCL, t.rank)
    Put(t.rank,
        elem_ptr)
    Unlock(t.rank)
    break
  }
  else {
    Lock(EXCL, t.rank)
    Get(t.rank,
        next_t)
    Unlock(t.rank)

    if (new_t.rank >= 0)
      t = next_t
  }
}
```

(a) MPI-2: Exclusive Lock/Unlock

```
for (;;) {
  if (t.rank == p-1) {
    Lock(SHR, t.rank)
    Acc(t.rank, REPLACE,
        elem_ptr)
    Unlock(t.rank)
    break
  }
  else {
    Lock(SHR, t.rank)
    Get_acc(t.rank, NOP,
        next_t)
    Unlock(t.rank)

    if (new_t.rank >= 0)
      t = next_t
  }
}
```

(b) MPI-3: Shared Lock/Unlock

```
for (;;) {
  if (t.rank == p-1) {

    Acc(t.rank, REPLACE,
        elem_ptr)
    Flush(t.rank)
    break
  }
  else {

    Get_acc(t.rank, NOP,
        next_t)
    Flush(t.rank)

    if (new_t.rank >= 0)
      t = next_t
  }
}
```

(c) MPI-3: Lock-All

Figure 9. Pseudocode for linked-list traverse-and-append loop in each idiom.

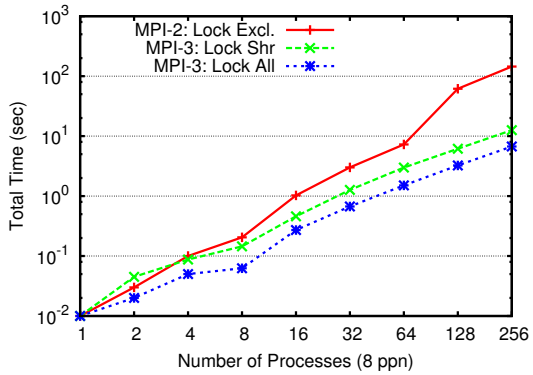


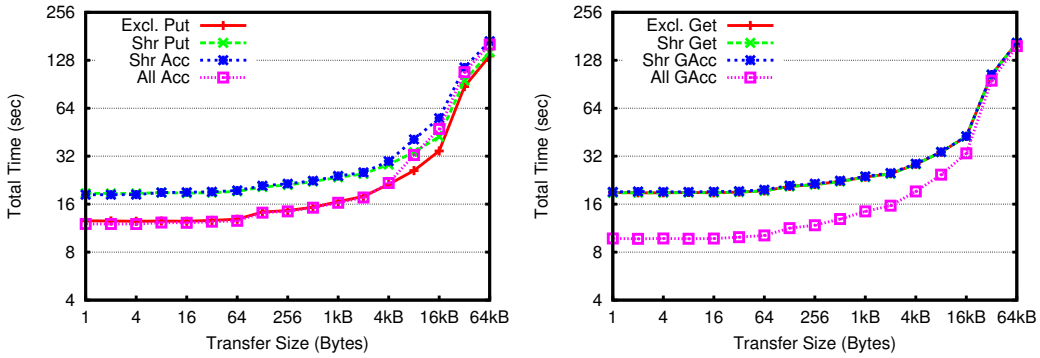
Figure 10. Distributed linked list creation and traversal time, for a $N = 10,000$ element list.

From this figure, we see that the MPI-2 exclusive lock communication mode provides the least tolerance in the presence of contention. In comparison, the MPI-3 implementations use the new accumulate operations that enable both to use a shared lock, providing greater concurrency and less overhead. At 256 processes, the MPI-3 implementation provides an more than an order of magnitude improvement in performance.

Comparing the MPI-3 shared lock and lock-all implementations, we see the additional protocol overhead reduction that is provided by the MPI-3 lock-all mode of operation. We measure this gap directly in Figure 11, which shows that lock-all provides a significant latency reduction by eliminating the communication involved in the lock operation. The corresponding protocol for each operation is shown in Figure 12. For the experiments in this section, we have disabled the lock-op-unlock merging optimization to provide a fair comparison, because it is not yet implemented for get-accumulate or flush operations.

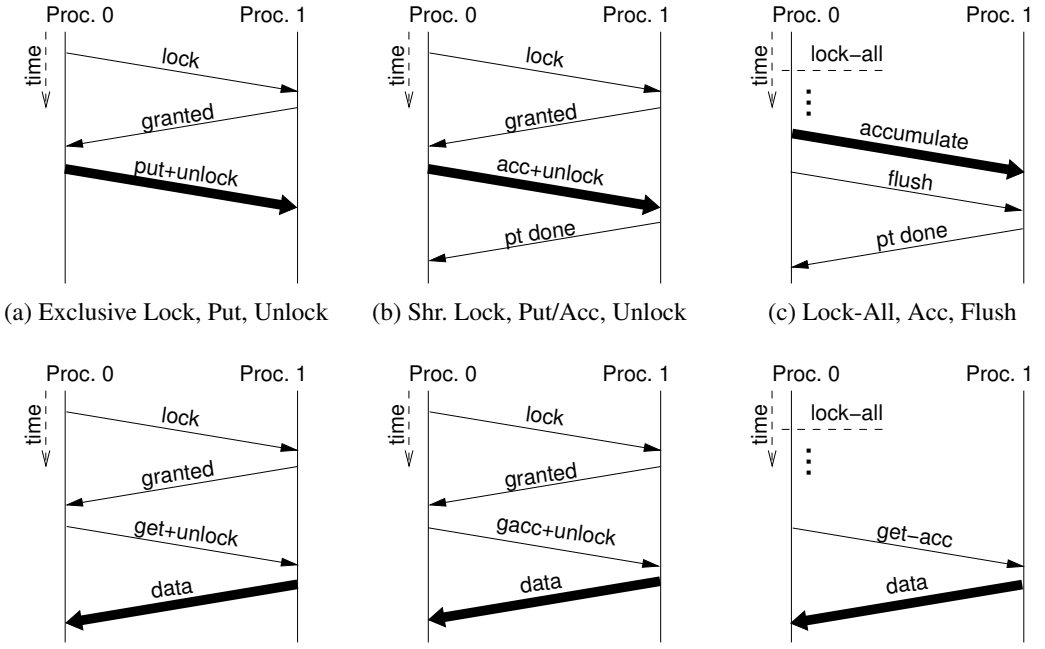
Comparing Figures 12a and 12b, we see that exclusive write epochs can have a lower protocol overhead that shared write epochs have. The added overhead in the case of shared access is to ensure remote completion, in case of third-party communication. When exclusive access epochs are used, the target ensures that all operations from an epoch are complete before granting access to another process. In Figure 10 ee see the benefit from the reduced protocol on up to four processes, but the serialization of exclusive access epochs quickly overcomes the performance advantages of the simpler protocol.

DOI: 10.1002/cpe



(a) Linked list write operation latency (b) Linked list read operation latency

Figure 11. Linked list read and write operation latency on the Eureka cluster.



(a) Exclusive Lock, Put, Unlock (b) Shr. Lock, Put/Acc, Unlock (c) Lock-All, Acc, Flush

(d) Exclusive Lock, Get, Unlock (e) Shr. Lock, Get/Get-Acc, Unlock (f) Lock-All, Get-Acc, Flush

Figure 12. Communication protocols used for linked-list pointer update and chase operations.

5.5. *Impact of Window Types*

Many applications generate and operate on data that mutates in size or layout during execution. To accommodate these types of algorithms, MPI-3 has added a dynamic window that allows processes to asynchronously add and remove memory to and from their window. In contrast, under MPI-2, windows are immutable and are created collectively. Thus, resizing a window required the programmer to collectively create a new window, copy data, and destroy the old window.

Figure 13 shows the time required to double the size of an RMA window using the MPI-2 approach and using MPI-3 dynamic windows. The figure shows traces for windows with an initial size ranging from 1kB to 16MB. For each window size, we vary the number of processes. Creating a window requires several collective operations, including an all-gather operation, whose overhead increases with the number of processes. In contrast, attaching memory to a dynamic window is a local operation with a fixed cost, regardless of the buffer sizes and the number of processes.

DOI: 10.1002/cpe

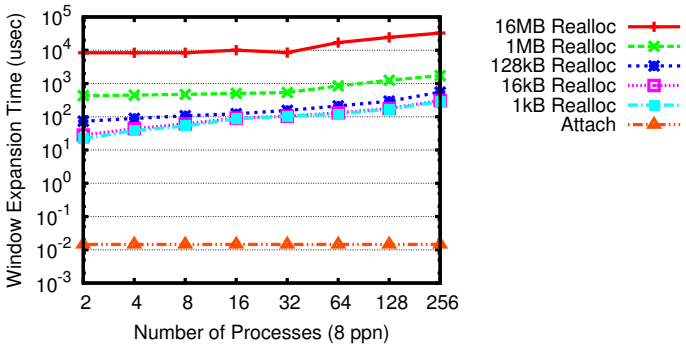


Figure 13. Time required to double the size of a window, by reallocating the window (MPI-2) versus attaching an additional buffer to a dynamic window (MPI-3).

6. RELATED WORK

A variety of low-level one-sided communication systems have been created, including Shmem [8], ARMCI [24], and GASNet [4]. These systems have been used independently and as runtime systems to support higher-level global address space models, such as Global Arrays [25], UPC [33], Co-Array Fortran v1.0 [26], Coarray Fortran v2.0 [19], Chapel [6], and X10 [7].

UPC implementers were unable to utilize MPI-2 RMA as a runtime system because of a semantic mismatch between MPI and UPC that could not be overcome at the runtime level [5]. Instead, an active message runtime was built on top of MPI two-sided messaging [3]. It is hoped that MPI-3 has addressed this gap and that it will be suitable as a low-level, portable runtime system for a variety of one-sided and global address space models.

The MPI-2 RMA interface has existed for over a decade, and significant effort has been invested in improving its performance [2, 12, 13, 16, 17, 21, 29, 31, 32, 35] and in building higher-level libraries using MPI RMA [18, 10]. MPI RMA has been demonstrated to be effective in a variety of applications, including earthquake modeling simulations [9] and cosmological simulations [30].

7. OUTLOOK AND FUTURE WORK

MPI-2 RMA defined a conservative but extremely portable system for one-sided communication. The MPI-2 memory model, termed the "separate" model in MPI-3 RMA, provided an efficient and portable interface on systems such as the Earth Simulator, then the world's fastest machine. However, the separate model did not exploit hardware that provided stronger guarantees about memory coherence (the MPI-3 "unified" model) or remote atomic operations; in addition, the completion models limited some of the common uses of one-sided programming. The enhancements provided by MPI-3 RMA have addressed these limitations and others and have resulted in a powerful, well-defined RMA model that fits within the MPI environment and can perform efficiently on current and future systems.

In this work, we have presented the first complete implementation of the MPI-3 RMA specification. While our implementation is feature complete, many opportunities for performance optimization and system integration still remain.

7.1. Management of RMA Communication and Synchronization Operations

A particularly important area for performance optimization is the management of RMA operations and synchronizations. Two key areas of future work are (1) piggybacking and merging synchronization messages with RMA operations and (2) efficiently managing RMA operations to optimize for short, latency-sensitive epochs, and long, bandwidth-bound epochs.

DOI: 10.1002/cpe

The design of MPI RMA synchronization gives the implementation great flexibility in deciding when to issue RMA operations. The most obvious approach is an *eager* approach, where operations are issued as they are encountered; and for large transfers this is often an efficient choice. However, for short updates such as a single word put or accumulate, this approach generates a significant amount of network traffic as well as large latencies while waiting for operations to complete. As shown in Figure 12, epoch synchronization operations can result in several additional messages.

An alternative *lazy* approach to synchronization queues all operations and waits until the unlock call to process them. When this approach is used, synchronization operations can be piggybacked, merged, or in some cases eliminated, yielding significant communication latency improvements. In the case of active target synchronization, it is possible to eliminate one of the barriers within the fence operations in exchange for a single reduce-scatter operation [12, 31]. The lazy approach is advantageous when epochs contain few operations and perform short data transfers. However, when there are either large numbers of RMA operations or the operations involve large amounts of data, it is often better to issue those operations as they are encountered. Thus, neither the eager nor lazy methods are always the best choice.

Until recently, MPICH implemented only the lazy synchronization method. We are currently implementing an adaptive method, described in [36], that can automatically switch from the lazy to the eager algorithm with low overhead, providing the benefits of both approaches transparently to the user. In addition, once both methods are implemented, it is easy to support user-supplied hints (e.g., through an "info" key when the MPI RMA window is created) on the type of data transfers that will occur.

7.2. Extensions for One-Sided Networks

The current messaging-based implementation of MPI RMA provides good performance for networks that do not natively support one-sided operations. However, many modern networks provide one-sided and RDMA support, which can yield considerable performance benefits.

We are currently investigating an extension to CH3 to better support devices that provide one-sided primitives. Such support would require the addition of function-pointers for network-supported one-sided operations to CH3's per connection data structure, the virtual channel structure. In addition, a channel can provide different function pointers per connection, depending on whether shared-memory or network communication should be used to perform one-sided accesses at the target. Significant challenges in this work will include the maintenance of data consistency and operation ordering when multiple communication mechanisms are used, for example, when put/get use RDMA, but long-double-precision accumulate requires the use of messaging.

REFERENCES

1. P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur. PMI: A scalable parallel process-management interface for extreme-scale systems. In *Proc. the Euro MPI Users' Group Conference (EuroMPI)*, Stuttgart, Germany, Sept. 12–15 2010.
2. Brian W. Barrett, Galen M. Shipman, and Andrew Lumsdaine. Analysis of implementation options for MPI-2 one-sided. In *Proc., Euro PVM/MPI*, 2007.
3. Dan Bonachea. AMMPI: Active messages over MPI. Website. `http://www.cs.berkeley.edu/~bonachea/ammpi/`.
4. Dan Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002.
5. Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *Int. J. High Perform. Comput. Netw.*, 1:91–99, August 2004.
6. B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *Intl. J. High Performance Computing Applications (IJHPCA)*, 21(3):291–312, 2007.
7. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Intl. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 519–538. ACM SIGPLAN, 2005.
8. SHMEM Community. OpenSHMEM specification v1.0. `http://www.openshmem.org`, 2012.
9. Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D.K. Panda, A. Chourasia, J. Levesque, S. M. Day, and P. Maechling. Scalable earthquake simulation on petascale supercomputers. In *Proceedings of the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, Nov 2010.

DOI: 10.1002/cpe

10. James Dinan, Pavan Balaji, Jeff R. Hammond, Sriram Krishnamoorthy, and Vinod Tipparaju. Supporting the global arrays PGAS model using MPI one-sided communication. In *Proc. 26th Intl. Parallel and Distributed Processing Symp.*, IPDPS '12, May 2012.
11. William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*, chapter 6.5.2. MIT Press, Cambridge, MA, 1999.
12. William Gropp and Rajeev Thakur. An evaluation of implementation options for MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*, pages 415–424. Lecture Notes in Computer Science 3666, Springer, September 2005.
13. William Gropp and Rajeev Thakur. Revealing the performance of MPI RMA implementations. In Franck Cappello, Thomas Herault, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 272–280. Springer, Berlin / Heidelberg, 2007.
14. Jeff Hammond, James Dinan, Pavan Balaji, Ivo Kabadshow, Sreeram Potluri, and Vinod Tipparaju. OSPRI: An optimized one-sided communication runtime for leadership-class machines. In *Proc. Sixth Conf. on Partitioned Global Address Space Programming Models*, PGAS '12, October 2012.
15. Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, Vivek Kale William Gropp, and Rajeev Thakur. Leveraging MPIs one-sided communication interface for shared-memory programming. In *Proc. Recent Adv. in MPI - 19th Euro. MPI Users Group Mtg.*, EuroMPI '12, September 2012.
16. Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, Darius Buntinas, Rajeev Thakur, and William Gropp. Efficient implementation of MPI-2 passive one-sided communication over InfiniBand clusters. In Dieter Kranzlmüller, Peter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting*, pages 68–76. Lecture Notes in Computer Science 3241, Springer, September 2004.
17. P. Lai, S. Sur, and D. K. Panda. Designing truly one-sided MPI-2 RMA intra-node communication on multi-core systems. In *International Supercomputing Conference (ISC)*, June 2010.
18. Robert Latham, Robert Ross, and Rajeev Thakur. Implementing MPI-IO atomic mode and shared file pointers using MPI one-sided communication. *Intl. J. of High Performance Computing Applications*, 21(2):132–143, 2007.
19. John Mellor-Crummey, Laksono Adhianto, William N. Scherer, III, and Guohua Jin. A new vision for coarray Fortran. In *Proc. of the Third Conf. on Partitioned Global Addr. Space Prog. Models*, PGAS '09. ACM, 2009.
20. John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.
21. Fernando Elson Mourão and João Gabriel Silva. Implementing MPI's one-sided communications for WMPI. In Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting*, pages 231–238. Lecture Notes in Computer Science 1697, Springer, September 1999.
22. MPI Forum. MPI-2: Extensions to the message-passing interface. Technical report, University of Tennessee, Knoxville, 1996.
23. MPI Forum. MPI: A message-passing interface standard version 3.0. Technical report, University of Tennessee, Knoxville, September 2012.
24. Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586, 1999.
25. Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, 2006.
26. Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.
27. C. Oehmen and J. Nieplocha. ScalaBLAST: A scalable implementation of BLAST for high-performance data-intensive bioinformatics analysis. *IEEE Trans. on Parallel and Distributed Systems*, 17(8):740 –749, aug. 2006.
28. R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen. Implementing MPI-IO atomic mode without file system support. In *IEEE Intl. Symp. on Cluster Comp. and the Grid*, CCGrid, May 2005.
29. G. Santhanaraman, P. Balaji, K. Gopalakrishnan, R. Thakur, W. Gropp, and D. K. Panda. Natively Supporting True One-sided Communication in MPI on Multi-core Systems with InfiniBand. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, Shanghai, China, May 18–21 2009.
30. Couchman H. M. P Thacker R. J., Pringle G. and S. Booth. HYDRA-MPI: An adaptive particle-particle, particle-mesh code for conducting cosmological simulations on mpp architectures. *High Performance Computing Systems and Applications*, 2003.
31. Rajeev Thakur, William Gropp, and Brian Toonen. Optimizing the synchronization operations in MPI one-sided communication. *Intl. J. High-Performance Computing Applications*, 19(2):119–128, Summer 2005.
32. Jesper Larsson Träff, Hubert Ritzdorf, and Rolf Hempel. The implementation of MPI-2 one-sided communication for the NEC SX-5. In *Proc. SC2000: High Performance Networking and Computing*, November 2000.
33. UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.
34. M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.
35. Joachim Worringen, Andreas Gäer, and Frank Reker. Exploiting transparent remote memory access for non-contiguous and one-sided-communication. In *Proc. 2002 Workshop on Communication Architecture for Clusters (CAC)*, April 2002.
36. Xin Zhao, Gopalakrishnan Santhanaraman, and William Gropp. Adaptive strategy for one-sided communication in MPICH2. In Jesper Träff, Siegfried Benkner, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 7490 of *Lecture Notes in Computer Science*, pages 16–26. Springer Berlin / Heidelberg, 2012.

DOI: 10.1002/cpe

Prepared using cpeauth.cls
DOI: 10.1002/cpe